Parallel Algorithms for Entropy-Coding Techniques [*]

ABDOU YOUSSEF
Department of EE & CS
The George Washington University
Washington, DC 20052, USA
email: youssef@seas.gwu.edu

Abstract: With the explosion of imaging applications, and due to the massive amounts of imagery data, data compression is essential. Lossless compression, also called entropy coding, is of special importance because not only it serves as a stand-alone system for certain applications such as medical imaging, it also is an inherent part of lossy compression. Therefore, fast entropy coding/decoding algorithms are desirable. In this paper we will develop parallel algorithms for several widely used entropy coding techniques, namely, arithmetic coding, run-length encoding (RLE), and Huffman coding. Our parallel arithmetic coding algorithm takes $O(\log^2 N)$ time on an N-processor hypercube, where N is the input size. For RLE, our parallel coding and decoding algorithms take $O(\log N)$ time on N processors. Finally, in the case of Huffman coding, the parallel coding algorithm takes $O(\log^2 N + n \log n)$, where n is the alphabet size, $n << N$. As for decoding, however, both arithmetic and Huffman decoding are hard to parallelize. However, special provisions could be made in many applications to make arithmetic decoding and Huffman decoding fairly parallel.

1 Introduction

With the explosion of imaging applications and due to the massive amounts of imagery data, data compression is essential to reduce the storage and transmission requirements of images and videos [3, 6, 8]. Compression can be lossless or lossy. Lossless compression, also called entropy coding, allows for perfect reconstruction of the data, whereas lossy compression does not. Even in lossy compression, which is by far more prevalent in image and video compression, entropy coding is needed as a last stage after the data has been transformed and quantized [8]. Therefore, fast entropy coding algorithms are of prime importance, especially in online or real-time applications such as video teleconferencing.

Parallel algorithms are an obvious choice for fast processing. Therefore, in this paper we will develop parallel algorithms for several widely used entropy coding techniques, namely, arithmetic coding [7], run-length encoding (RLE) [9], and Huffman coding [2]. Our parallel arithmetic coding algorithm takes $O(\log^2 N)$ time on an N-processor hypercube, where N is the input size. Unfortunately, arithmetic decoding seems to be hard to parallelize because it is a sequential process of essentially logical computations. In practice, however, files are broken down into many substrings before being arithmetic-coded, for precision reasons that will become clear later on. Accordingly, the coded streams of those substrings can be decoded in parallel.

For RLE, we design parallel algorithms for both encoding and decoding, each taking $O(\log N)$ time. Finally, in the case of Huffman coding, the coding algorithm is easily data-parallel. The statistics gathering for computing symbol probabilities before constructing the Huffman tree is parallelized to take $O(\log^2 N)$ time. Like arithmetic coding, Huffman decoding is highly sequential. However, in certain applications where the data is inherently broken into many blocks that are processed independently as in JPEG/MPEG [3, 6], simple provisions can be made to have the bitstreams easily separable into many independent substreams that can be decoded independently in parallel.

It must be noted that other lossless compression techniques are also in use such as Lempel-Ziv, bitplane coding, and differential pulse-code modulation (DPCM) [8]. The first two will be considered in future work. The last technique, DPCM, is the subject matter of another paper appearing in this conference [10].

The paper is organized as follows. The next section gives a brief description of the various standard parallel operations that will be used in our algorithms. Section 3 develops a parallel algorithm for arithmetic coding. Section 4 develops parallel encoding and decoding algorithms for RLE. Section 5 addresses the parallelization of Huffman coding and decoding. Conclusions and future directions are given in section 6.

2 Preliminaries

The parallel algorithms designed in this paper use several standard parallel operations. A list of those operations along with a brief description will follow.

- **Parsort**($Y[0 : N-1]$; $Z[0 : N_1]$, $\pi[0 : N-1]$): This operation sorts in parallel the input array Y into the output array Z, and records the permutation π that orders Y to Z: $Z[k] = Y[\pi[k]]$. Of the many parallel sorting algorithms, we use the fastest practical one, namely, Batcher's bitonic sorting [1], which takes $O(\log^2 N)$ parallel time on an N-processor hypercube.

- C=**Parmult**($A_{0:N-1}$): It multiplies the N elements of the array A, yielding the product C. In

[*]This work was performed in part at the National institute of Standards and technology.

this paper, the elements of A are 2×2 matrices. This operation clearly takes simply $O(logN)$ time on $O(N)$ processors connected as a hypercube.

- $A[0, N-1]$=**Parprefix**$(a_{0:N-1})$: This is the well-known parallel prefix operation [5]. It computes from the input array a the array A where $A[i] = a[0] + a[1] + ... + a[i]$, for all $i = 0, 1, ..., N-1$. Parallel prefix takes $O(\log N)$ time on N processors connected in a variety of ways, including the hypercube.

- $A[0 : N-1]$=**Barrier-Parprefix**$(a[0 : N-1])$: This operation assumes that the input array a is divided into groups of consecutive elements; every group has a *left-barrier* at its start and a *right-barrier* at its end. **Barrier-Parprefix** performs a parallel prefix within each group independently from other groups. **Barrier-Parprefix** takes $O(\log N)$ time on an N-processor hypercube. To see this, let $f[0 : N-1]$ be a flag array where $f[k] = 0$ if k is a right-barrier, and $f[k] = 1$, otherwise. Clearly, $A[i] = f[i-1]A[i-1] + a[i]$ for all $i = 0, 1, ..., N-1$. The latter is a linear recurrence relation which can be solved in $O(\log N)$ time on an N-processor hypercube [4].

3 Parallel Arithmetic Coding

Arithmetic coding [7] relies heavily on the probability distributions of the input files to be coded. Essentially, arithmetic coding maps each input file to a subinterval $[L\ R]$ of the unit interval $[0\ 1]$ such that the probability of the input file is $R - L$. Afterwards, it represents the fraction value L in n-ary using $r = \lceil -\log_n(R-L) \rceil$ n-ary digits, where n is the size of the alphabet. The stream of those r digits are taken to be the code of the input file.

The mapping of an input file into a subinterval $[L\ R]$ is done progressively by reading the file and updating the $[L\ R]$ subinterval to always be the corresponding subinterval of the input substring scanned so for. The update rule works as follows. Assume that the input file is the string $x[0 : N-1]$ where every symbol is in the alphabet $\{a_0, a_1, ..., a_{n-1}\}$, and that the substring $x[0 : k-1]$ has been processed, i.e., mapped to interval $[L\ R]$. Let P_{ki} be the probability that the next symbol is a_i given that the previous symbols are $x[0 : k-1]$. Divide the current interval $[L\ R]$ into n successive subintervals where the i-th subinterval is of length $P_{ki}(R-L)$ for $i = 0, 1, ..., n-1$, that is, the i-th subinterval is $[L_i\ R_i]$ where $L_i = (P_{k0} + P_{k1} + ... + P_{k,i-1})(R-L)$ and $R_i = L_i + P_{ki}(R-L)$. Finally, if the next symbol in the input file is a_{i_0} for some i_0, the update is $L = L_{i_0}$ and $R = R_{i_0}$. The last value of the interval $[L\ R]$ after the whole input string has been processed is the desired subinterval.

The alphabet $\{a_0, a_1, ..., a_{n-1}\}$ can be arbitrary. Some of the typical alphabets are the binary alphabet $\{0, 1\}$ for binary input files, the ascii alphabet, and any finite set of real numbers or integers as may occur in run-length encoding. In the last category, the alphabet $\{a_0, a_1, ..., a_{n-1}\}$ can be easily mapped to the more convenient alphabet $\{0, 1, ..., n-1\}$. That mapping is applied at the outset before arithmetic coding starts, and the inverse mapping is applied after arithmetic decoding is completed. Henceforth, we will assume the alphabet to be $\{0, 1, ..., n-1\}$.

The conditional probabilities $\{P_{ki}\}$ are either computed statistically from the input file or derived from an assumed theoretical probabilistic model about the input files. Naturally, the statistical method is the one used most often, and will be assumed here. The structure of the probabilistic model is, however, still useful in knowing what statistical data should be gathered. The model often used is the Markov model of a certain order m, where m tends to be fairly small, in the order of 1–5. That is, the probability that the next symbol is of some value a depends on only the values of the previous m symbols. Therefore, to determine statistically the probability that the next symbol is a given that the previous m symbols are some $b_1 b_2 ... b_m$, it suffices to compute the frequency of occurrences of the substring $b_1 b_2 ... b_m a$ in the input string, and normalize that frequency by N, which is the total number of substrings of length $m+1$ symbols in the zero-padded input string. The padding of m 0's to the left of x is taken to simplify the statistics gathering at the left boundary of x: assume that the imaginary symbols $x[-m : -1]$ are all 0.

To summerize, the sequential algorithm for computing the statistical probabilities and performing arithmetic coding is presented next.

```
Algorithm Arithmetic-coding(in: $x[0 : N-1]$; out: $B$)
begin
/* The alphabet is assumed to be $\{0, 1, ..., n-1\}$ */
  Phase I: Statistics Gathering
  for $k = 0$ to $N-1$ do
     /* compute $\{P_{ki}\}'s$ which are initialized to 0*/
     compute the frequency $f_k$ of the substring
     $x[k-m : k]$ in the whole string $x$;
     set $Q_k = f_k/N$;
     let $i = x[k]$, and set $P_{ki} = Q_k$;
  endfor
  for $k = 0$ to $N-1$ do
     Let $i = x[k]$; set $P_k = P_{k0} + P_{k1} + ... + P_{k,i-1}$;
  endfor

  Phase II: finding the interval $[L\ R]$ correspond-
  ing to the string $x$
  Initialize: $L = 0$ and $R = 1$;
  for $k = 0$ to $N-1$ do
     $D = R - L$; $L = L + P_k D$; $R = L + Q_k D$;
  endfor

  Phase III: computing the output stream $B$
  $r = \lceil -\log_n(R-L) \rceil$;
  Take the $n$-ary representation of $L = 0.L_1 L_2 ...$;
  $B = [L_1\ L_2\ ...\ L_r]$;
end
```

To parallelize the Arithmetic-coding algorithm, the first two phases have to be parallelized. Note that in Phse III, L is naturally represented in binary inside the computer, so that phase is nothing more than computing $\lceil r \log n \rceil$ and chopping off the first r bits of the binary representation of L.

Parallelization of Phase I: Statistics Gathering

Each substring $x[k-m:k]$ is treated as an $(m+1)$-tuple of integer components, for $k = 0, 2, ..., N-1$; those N tuples are denoted as an array $Y[0 : N-1]$. Denote the $m+1$ components of $Y[k]$ as $(Y_m[k], ..., Y_1[k], Y_0[k])$, that is, $Y_0[k] = x[k]$, $Y_1[k] = x[k-1]$, $\ldots$, $Y_m[k] = x[k-m]$. Sort Y into Z using **Parsort**$(Y; Z, \pi)$, where $Z[k] = Y[\pi[k]]$. Clearly, all identical tuples are consecutive in Z. Associate $Q_{\pi[k]}$ and $P_{\pi[k],Z_0[k]}$ with tuple $Z[k] = Y[\pi[k]]$.

We will divide Z into segments and supersegments. A *segment* of value T is a maximal subarray of consecutive elements of Z where every tuple is of value T. Clearly, Z has as many segments as there are distinct tuples in Y. A *supersegment* is a maximal set of consecutive segments where the tuple value differ in only the rightmost component. Note that there can be at most n segments per supersegment because there are only n different alphabetic values for the rightmost component of a tuple. The probabilities Q_k's and $P_{k,x[k]}$'s are then computed as follows:

```
Procedure Compute-probs(input: $Z[0:N-1], \pi[0:N-1]$; output: $Q_k$'s, $P_{k,x[k]}$'s)
begin
  1. Put a left-barrier and a right-barrier at the beginning and at the end of every segment, respectively. It can be done in the following way. First, put a left barrier at $k=0$ and a right barrier at $k=N-1$. Afterwords, do
     for $k = 0$ to $N-2$ pardo
         if $Z[k] < Z[k+1]$, put a right barrier at $k$ and a left barrier at $k+1$.
     endfor
  2. Let $g[0:N-1]$ be an integer array where every term is initialized to 1;
  3. $G[0:N-1]$=Barrier-Parprefix($g$).
     Clearly, if $k$ corresponds to a right barrier of a segment, then $G[k]$ is the number of terms of that segment, that is, $G[k]$ is the frequency of $Z[k] = Y[\pi[k]]$.
  4. Broadcast within each segment the $G[k]$ of the segment's right barrier, and then set in parallel every $G[i]$ term in the segment to $G[k]$.
  5. for $k = 0$ to $N-1$ pardo
         set $Q_{\pi[k]} = G[k]/N$ and $P_{\pi[k],Y_0[\pi[k]]} = G[k]/N$.
     endfor
end
```

Observe that the $Q_{\pi[k]}$'s within any one single segment, and therefore the $P_{\pi[k],Y_0[\pi[k]]}$'s within any segment, are all equal. We call $Q_{\pi[k]}$ (or $P_{\pi[k],Y_0[\pi[k]]}$) the probability of that segment. Observe also that the cumulative probability P_k, which is defined as $P_{k0} + P_{k1} + ... + P_{k,x[k]-1}$, is the sum of probabilities of all tuples where the m leftmost symbols are equal to $x[k-m : k-1]$ and where the rightmost symbol is $\leq x[k]-1$. Stated otherwise, $P_{\pi[k]}$ is the sum of the probabilities of all the segments within the supersegment containing k such that the m leftmost symbols are equal to those of $Y[\pi[k]]$ and the rightmost symbol is $\leq x[\pi[k]]-1$. The following procedure will compute those cumulative probabilities P_k's.

```
Procedure Compute-cumprobs(input: $Z$, $\pi$, $Q_k$'s, $P_{ki}$'s; output: $P_k$'s)
begin
  1. Put a left-barrier and a right-barrier at the beginning and at the end of every supersegment, respectively, as follows. For each $k$, denote by $Z'[k]$ the $m$-tuple consisting of the $m$ leftmost components of $Z[k]$, that is, $Z'[k]$ is all but the rightmost component of $Z[k]$. Put a left barrier at $k=0$ and a right barrier at $k=N-1$. Afterwords, do
     for $k = 0$ to $N-2$ pardo
             if $Z'[k] < Z'[k+1]$, put a right barrier at $k$ and a left barrier at $k+1$.
     endfor
  2. Let $h[0:N-1]$ be a real array where every term is initialized to 0; for each $k = 0, 1, 2, ..., N-1$, $h[k]$ is associated with $Z[k]$.
     for $k = 0$ to $N-1$ pardo
         if $k$ happens to be the start of a segment (rather than a supersegment), then
        set $h[k] = P_{\pi[k],Z_0[k]}$.
     endfor
  3. $H[0:N-1]$=Barrier-Parprefix($h$), using the supersegment barriers.
     Note that $H[k]$ = the sum of the probabilities of all the segments within the supersegment containing $k$ such that the $m$ leftmost symbols are equal to those of $Y[\pi[k]]$ and the rightmost symbol is $\leq x[\pi[k]]$. After the discussion above, $H[k] = P_{\pi[k]} + Q_{\pi[k]}$. This justifies the next step.
  4. for $k = 0$ to $N-1$ pardo
             $P_{\pi[k]} = H[k] - Q_{\pi[k]}$.
     endfor
end
```

Time Analysis

For each k, assume that $x[k]$, $Y[k]$ and $Z[k]$ will be hosted by processor k. The gathering of $x[k-m:k]$ to processor k to form $Y[k]$ requires m shifts that send data from node i to node $i+1$ for all i. Each shift takes $O(\log N)$ communication time on an N-processor hypercube. Therefore, the forming of Y takes $O(m \log N) = O(\log N)$ communication time for the m shifts. The reason m was dropped from the time formula is because m is fairly small, in the order of $1-5$ usually, and thus is assumed to be a constant.

Parsort takes $O(\log^2 N)$ time on an N-processor hypercube.

Procedure Compute-probs will be shown to take $O(\log N)$ time. Step 1 involves an exchange of the values $Z[k]$ and $Z[k+1]$ between processors k and $k+1$, for all k. This is accomplished by two shifts: one from k to $k+1$ and the other from $k+1$ to k, for all k. Thus, this step takes $O(\log N)$ time. Step 2 takes $O(1)$ time. Step 3, **Barrier-Parprefix**, takes $O(\log N)$ time. Step 4, being several independent broadcasts within nonoverlapping portions of the hypercube, also takes $O(\log N)$ time. Finally, step 5 takes $O(1)$ time because it is a simple parallel step. This establishes that the whole procedure takes $O(\log N)$ parallel time.

The analysis of the procedure Compute-cumprobs is very similar, and shows that it takes $O(\log N)$ parallel time as well.

It must be noted that after executing the last two procedures, the probabilities P_k and Q_k are to be sent to processor k, for each $k = 0, 1, ..., N-1$. At present, $P_{\pi[k]}$ and $Q_{\pi[\pi[k]}$ are in processor k along with $Z[k]$. Therefore, for all k, processor k sends $P_{\pi[k]}$ and $Q_{\pi[\pi[k]}$ to processor $\pi[k]$. That is, this communication step is just a permutation routing of π. If routed using Valiant's randomized routing algorithm, it will take $O(\log N)$ communication time with overwhelming probability. Otherwise, π can be routed by bitonic sorting of its destinations, taking $O(\log^2 N)$ time.

In conclusion, the statistics gathering process takes $O(\log^2 N)$ parallel time for both communication and computation. It remains to parallelize Phase II of arithmetic coding.

Parallelization of Phase II: the Computation of $[L\ R]$

It will be shown that the computation of the interval $[L\ R]$ is the product of N 2×2 matrices formed from the probabilities P_k and Q_k. Afterwards, we can use the parallel operation **Parmult** to multiply the N matrices in $O(\log N)$ time on N processors.

Let the updated values of L and R at iteration k of the for-loop of Phase II be denoted L_k and R_k, respectively. Clearly, $L_k = L_{k-1} + P_k(R_{k-1} - L_{k-1}) = (1-P_k)L_{k-1} + P_k R_{k-1}$, and $R_k = L_k + Q_k(R_{k-1} - L_{k-1}) = (1-P_k)L_{k-1} + P_k R_{k-1} + Q_k(R_{k-1} - L_{k-1}) = (1 - P_k - Q_k)L_{k-1} + (P_k + Q_k)R_{k-1}$. In summary, we have

$$L_k = (1-P_k)L_{k-1} + P_k R_{k-1},$$

$$R_k = (1 - P_k - Q_k)L_{k-1} + (P_k + Q_k)R_{k-1}. \quad (1)$$

Letting

$$X_k = \begin{bmatrix} L_k \\ R_k \end{bmatrix}, \quad A_k = \begin{bmatrix} 1-P_k & P_k \\ 1-P_k-Q_k & P_k+Q_k \end{bmatrix}, \quad (2)$$

equation 1 becomes a simple vector recurrence relation of order 1:

$$X_k = A_k X_{k-1}. \quad (3)$$

The last equation implies that the last subinterval $[L\ R] = [L_{N-1}\ R_{N-1}]$ that is being sought, which corresponds to X_{N-1}, is

$$X_{N-1} = A_{N-1}A_{N-2}\cdots A_0 X_{-1},$$

or equivalently,

$$\begin{bmatrix} L_{N-1} \\ R_{N-1} \end{bmatrix} = A_{N-1}A_{N-2}\cdots A_0 \begin{bmatrix} L_{-1} \\ R_{-1} \end{bmatrix}. \quad (4)$$

Since $X_{-1} = [L_{-1}\ R_{-1}]^t = [0\ 1]^t$, it follows that $[L_{N-1}\ R_{N-1}]^t$ is the right column of the product matrix $A_{N-1}A_{N-2}\cdots A_0$. That product is clearly computable with the parallel operation **Parmult**$(A_{N-1:0})$, taking $O(N/P + \log P)$ matrix multiplications on P ($\leq N$) processors. Noting that the matrices are 2×2, each matrix multiplication takes $O(1)$ time on a single processor, and therefore, the computation of the final interval $[L\ R]$ in Phase II takes $O(\log N)$ parallel time on N processors.

At this point, the parallel algorithm for arithmetic coding can be put together as follows.

```
Algorithm Par-arithm-coding(in: $x[0:N-1]$; out: $B$)
begin
    Form the array $Y[0:N-1]$ of tuples;
    Parsort($Y$; $Z, \pi$);
    Compute-probs($Z, \pi$; $Q_k$'s, $P_{k,x[k]}$'s);
    Compute-cumprobs($Z$, $\pi$, $Q_k$'s, $P_{k,x[k]}$'s; $P_k$'s);
    Route $\pi$ to send $P_{\pi[k]}$ and $Q_{\pi[k]}$ to processor $k$, $\forall k$;
    for $k = 0$ to $N-1$ pardo
        $A_k = \begin{bmatrix} 1-P_k & P_k \\ 1-P_k-Q_k & P_k+Q_k \end{bmatrix}$;
    endfor
    $C =$ Parmult($A_{N-1:0}$);
    $L = C(1,2)$; $R = C(2,2)$;
    $r = \lceil -\log_n(R-L) \rceil$;
    Take the $n$-ary representation of $L = 0.L_1L_2...$;
    $B = [L_1\ L_2\ ...\ L_r]$;
end
```

Time of the whole algorithm: Based on the preceding time analyses, the overall parallel time of the algorithm is $O(\log^2 N)$ on an N-processor hypercube. Indeed, the parallel sorting is what dominates the time, for otherwise, the algorithm takes $O(\log N)$ time.

Arithmetic Decoding

Arithmetic decoding, which reconstructs the string (or x) from the stream B, is much harder to parallelize. It works as follows, assuming the probabilities are available for simplicity. The interval $[L\ R]$ is narrowed down progressively as in coding, where the initial value is $[0\ 1]$. The final interval, call it $[L_f, R_f]$, is known at decoding time from the bit stream: $D = n^r$ where r is the length of the stream B, $L_f =$ (stream B as an n-ary number)$/n^r$, and $R_f = L_f + D$. To figure out the next symbol in the file, using the next n-ary digit $B[i]$ in the stream B, the current interval $[L\ R]$ is divided into n subintervals as in coding, one subinterval per alphabet symbol; afterwards, decode $B[i]$ as alphabet symbol a_j if $[L_f\ R_f]$ is contained within the j-th subinterval. Thus, the recurrence relation for the decoded symbols involves essentially positional rather than numerical computations, making it hard to parallelize its computation.

In practice, however, arithmetic coding is applied in a way that allows for some decoding parallelism. Because of accuracy problems, if the input string size N is fairly large, the intermediary intervals $[L\ R]$ become too small for the precision afforded by most computers. Therefore, long input files are broken into several substrings of acceptable lengths; those substrings are arithmetic-coded independently, except perhaps in the statistics gathering, which involves the whole file to reduce the probability model information overhead to be included in the header of the stream B. Accordingly, the streams of those substrings can be decoded independently **in parallel**. The actual details are not included here, and will vary from application to application, although the principle is the same.

4 Parallel Run-Length Encoding

Run-length encoding (RLE) [9] applies with good performance when the input string $x[0:N-1]$ consists of a relatively short sequence of runs (say r runs, $r << N$), where a run is a substring of consecutive symbols of equal value. RLE converts x into a sequence of pairs (L_0, V_0), (L_1, V_1), ... ,(L_{r-1}, V_{r-1}), where L_i is the length of the i-th run, and V_i is the value of the recurring symbol of that run.

Often, there is redundancy in the values of the L_i's and the V_i's. In that case, Huffman coding [2] is applied to code the L_i's and/or the V_i's. Parallelizing Huffman coding is the subject of the next section. However, in the parallel RLE algorithm, we will put the data in the right form and location, and will gather the necessary statistics needed for Huffman coding.

The parallel RLE algorithm coincides with the first 3 steps of the algorithm 'Compute-probs' that was developed earlier for arithmetic coding. The segments there correspond to runs in RLE. After those steps execute, each right barrier of a segment has the L and V of its run. Afterwards, the scattered (L, V) pairs should be gathered to the first r processors in the system, in case further processing is needed, as for example Huffman coding the L_i's and the V_i's. The parallel algorithm for RLE can now be given as follows.

```
Algorithm Par-RLE(in: $x[0:N-1]$; out: $L$, $V$)
begin
1.  Put a left-barrier at $k=0$ of $x$, and a right-barrier
    at $k=N-1$;
2.  for $k=0$ to $N-2$ pardo
3.      if $x[k] \neq x[k+1]$ then
            put a left-barrier at $k$ and a right-
            barrier at $k+1$;
    endfor

4.  Let $g[0:N-1]$ be an integer array where every
    term is initialized to 1;
5.  $G[0:N-1]$=Barrier-Parprefix($g$);
    /* if $k$ is a right-barrier, $G[k]$ is the length of the
    corresponding run */

6.  Let $h[0:N-1]$ be an array initialized to 0;
7.  for $k=0$ to $N-1$ pardo
8.      if $k$ is a right-barrier, set $h[k]=1$;
    endfor
9.  $H[0:N-1]$=Barrier-Parprefix($h$);
    /* when $k$ is a right-barrier and $i=H[k]-1$, the
    corresponding run is the $i$-th run of $x$*/

10. for $k=0$ to $N-1$ pardo
11.     if $k$ is a right-barrier then
12.         $i=H[k]-1$; $L_i=G[k]$; $V_i=x[k]$;
13.         Processor $k$ sends $(L_i, V_i)$ to processor $i$;
        endif
    endfor
end
```

Time Analysis of Par-RLE

The systems is assumed to be an N-processor hypercube. Steps 1-3 for barrier setting take $O(1)$ parallel time. Steps 4-5 take $O(\log N)$ parallel time. Steps 6-8 take $O(1)$ parallel time, and step 9, being **Barrier-Parprefix** like step 5, takes $O(\log N)$ parallel. The computation in steps 10-12 takes $O(1)$ parallel time, while the communication in those steps, which is essentially a partial-permutation routing, takes $O(\log N)$ time using Valiant's randomized routing algorithm. Therefore, the whole algorithm takes $O(\log N)$ parallel time.

Parallel Run-length Decoding

It remains to parallelize run-length decoding (RLD). It is reasonable to start from the L_i's and V_i's as input, where L_i and V_i are in processor i, for $i = 0, 1, ..., r-1$. The RLD algorithm is supposed to reconstruct the original string x, where the first L_0 symbols are all V_0, the next L_1 symbols are all V_1, and so on. The algorithm must determine the start location and end location of each run i. Clearly, the first run start at location 0, the second run at location L_0, the next run at location $L_0 + L_1$, and so on. In other terms, for all $i = 1, 2, ..., r-1$, run i starts at location $L_0 + L_1 + ... + L_{i-1}$, and ends at location $L_0 + L_1 + ... + L_{i-1} + L_i - 1$, while run 0 starts at 0 and ends at $L_0 - 1$. This dictates the use of **Parprefix** to compute all those prefix sums of L in $O(\log r)$ parallel time on r processors. Afterwards, for $i = 1, 2, ..., r-1$, processor i must send (L_i, V_i) to processor $L_0 + L_1 + ... + L_{i-1}$; the sending of those $r-1$ message is a partial-permutation routing that takes $O(\log N)$ time on the hypercube. Finally, those recipients (processors) of the (L_i, V_i)'s, including processor 0 which has (L_0, V_0), broadcast in parallel their value V_i to the next L_i processors. Those r broadcasts are independent and run in nonoverlapping parts of the hypercube, thus taking $O(max(\{\log L_i\}) = O(\log N)$ communication time. This complete the decoding, which clearly takes on the whole $O(\log N)$ parallel time.

5 Parallel Huffman Coding

In Huffman coding the individual symbols of the alphabet are coded in binary using the frequencies (or probabilities) of occurrences of the symbols, such that no symbol code is the prefix of another symbol code. Afterwards, the input file (or string $x[0:N-1]$) is coded by replacing each symbol $x[i]$ by its code.

The Huffman coding algorithm for coding the alphabet is a greedy algorithm and works as follows. Suppose that the alphabet is $\{a_0, a_1, ..., a_{n-1}\}$, and let p_i be the probability of occurrence of symbol a_i, for $i = 0, 1, ..., n-1$. A Huffman binary tree is built by the algorithm. First, a node is created for each alphabet symbol; afterwards, the algorithm repeatedly selects two unparented nodes of smallest probabilities, creates a new parent node for them, and makes the probability of the new node to be the sum of the probabilities of its two children. Once the root is created, the edges of the tree are labeled, left edges with 0, right edges with 1. Finally, each symbol is coded with the binary sequence that labels the path from the root to the leaf node of that symbol. By creating a min-heap for the original leaves (according to the probabilities), the repeated insertions and deletions on the heap will take

$O(n \log n)$ time. The labeling of the tree and extractions of the leaf codes take $O(n)$ time. Therefore, the whole algorithm for alphabet coding takes $O(n \log n)$ time. Considering that the alphabet tends to be very small in size, and independent of the — much larger — size of the input files to be coded, $O(n \log n)$ is relatively very small.

This process of computing the probabilities p_i's is parallelizable as was done in the previous two sections: sort the input string in parallel using **Parsort**, then use **Barrier-Parprefix** to compute the frequencies of the distinct symbols in the input string. Those frequencies are then divided by N to obtain the probabilities, although this step is unnecessary since Huffman coding would give the same results if it uses frequencies instead of probabilities. The statistics gathering process clearly takes $O(\log^2 N)$ parallel time.

Once the symbol codes have been determined, each symbol $x[i]$ is replaced by its code, and all symbols are so processed in parallel. The concatenation of all the symbol codes is the output bitstream. This code replacement process takes $O(1)$ parallel time, since the length of each symbol code is $\leq n$ and is thus a constant. In summary, the total time of Huffman coding an input file of N symbols is $O(n \log n + \log^2 N)$.

Huffman decoding works as follows, assuming that the Huffman tree is available. The bitstream is scanned from left to right. When a bit is scanned, we traverse the Huffman tree one step down, left if the bit is 0, right if the bit is 1. Once a leaf is encountered, the scanned substring that led from the root to the leaf is replaced (decoded) by the symbol of that leaf. The process is repeated by resetting the traversal to start from the root, while the scanning continues from where it left off. Clearly, this decoding process is very hard to parallelize, and it may be inherently sequential. No attempt is made here to prove that.

One approach can be followed to bring some parallelism into Huffman decoding. In many applications and compression standards such as JPEG, MPEG2, and the upcoming MPEG4, the data is divided into blocks at some stage in the compression process, and the blocks are then quantized then entropy-coded independently of one another. The bitstreams of those blocks are then concatenated into a single bit stream according to some static ordering scheme of the blocks. A special End-of-Block (EOB) symbol is added to the alphabet and entropy-coded like other symbols; the EOB symbol tells the decoder when a block ends and the next begins. If parallelization is needed, the bitstreams of the various blocks should NOT be concatenated into one single stream. Rather, they should be formed into as many separate streams as there are processors to be used for decoding. That way, the separate streams can be decoded independently in parallel. By attempting to the many streams to be of roughly equal length, the decoding processes could be load balanced, leading to nearly optimal parallel decoding. The details of that approach, and the actual structure of the file that contains the separate bitstreams, are left to future work.

6 Conclusions

In this paper we developed parallel algorithms for several widely used entropy coding techniques, namely, arithmetic coding, run-length encoding, and Huffman coding. In all three, the coding turned out to be parallelizable, taking mainly $O(\log N)$ time on N processors, except in the cases where sorting was used for statistics gathering, requiring $O(\log^2 N)$ time. Decoding, however, turned out to be much harder to parallelize, except in the RLE case which is logarithmic in time. In practice, However, both arithmetic and Huffman coding are used in such a way that allows for simple parallel decoding.

References

[1] K. E. Batcher, "Sorting Networks and their Applications," *1968 Spring Joint Comput. Conf., AFIPS Conf.* Vol. 32, Washington, D.C.: Thompson, 1968, pp. 307–314.

[2] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proc. IREa,* Vol. 40, pp. 1098–1101, 1962.

[3] "ISO-13818-2: Generic Coding of Moving Pictures and Associated Audio (MPEG-2)."

[4] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.,* C-22, No. 8, pp. 786–793, Aug. 1973.

[5] C. P. Kruskal, L. Rudolph and M. Snir, "The power of parallel prefix," *IEEE Trans. Comput.'* Vol. 34, pp. 965–968, 1985.

[6] B. Pennebaker and J. L. Mitchell, *JPEG Still Image Data Compression Standard,* Van Nostrand Reinhold, New York, 1993.

[7] J. Rissanen and G. Langdon, "Arithmetic Coding," *IBM J. Res. Develop.* 23, pp. 149–162, March 1979. Also in *IEEE Trans. Comm.* COM-29, No. 6, pp. 858–867, June 1981.

[8] K. Sayood, *Introduction to Data Compression,* Morgan Kauffmann Publishers, San Fransisco, California, 1996.

[9] H. Tanaka and A. Leon-Garcia, "Efficient Run-Length Encoding," *IEEE Trans. Info. Theory,* IT-28, No. 6, pp. 880–890, 1987.

[10] A. Youssef, "An Efficient Algorithm for Multi-indexed Recurrence Relations with Applications to Compression of Multidimensional Signals," *IASTED Int'l Conference on Parallel and Distributed Systems,* Vienna, Austria, July 1998.